Procedural Guard Placement for Stealth Games

Qihan Xu
School of Computer Science
McGill University, Montréal
Québec, Canada
qihan.xu@mail.mcgill.ca

Jonathan Tremblay
School of Computer Science
McGill University, Montréal
Québec, Canada
jtremblay@cs.mcgill.ca

Clark Verbrugge
School of Computer Science
McGill University, Montréal
Québec, Canada
clump@cs.mcgill.ca

ABSTRACT

Stealth game mechanics rely on a suitably difficult distribution of enemy observers, the placement of which is typically a manual process. Here we investigate an automatic process for placement of observer opponents. We use a Monte-Carlo approach to generate randomized enemy positions and motions and combine this with a stealth path-planning and analysis framework. This allows us to ensure feasibility of the level design, and also measure relative difficulty. Initial results using this process compare placement of both mobile and static guards (rotating cameras), and let us explore the impact on level difficulty produced by different kinds of enemy observer agents.

Keywords

Computer games, stealth, procedural generation

1. INTRODUCTION

Stealth games require a player to traverse an area while avoiding detection by enemy agents. This task is made more or less difficult depending on the position and behaviours of guards, cameras, and other level components that positively or negatively affect a player's ability to sneak. Ensuring a stealth problem is both feasible and challenging, however, is itself a challenging task, and automated approaches that reduce the need for manual design and labour-intensive playtesting have obvious value.

We present a simple approach to automating guard placement in a stealth game context. Our design is based on first decomposing a level-space into Voronoi regions, and then using that region geometry to define appropriate observer locations, selecting either basic patrol paths for mobile guards, or static locations for rotational camera positioning. This design is composed with an existing tool for analyzing the existence of stealth paths. In this way we can automatically verify feasibility, and evaluate the resulting level difficulty—a level that admits more solutions is heuristically easier for players to solve than one that admits just a few. The overall process we describe is iterative and stochastic, allowing for high variability in the generated output, scalable time complexity, and easy integration into an iterative design process.

Specific contributions include:

- We describe an overall workflow along with basic algorithms for mobile guard and camera placement in a 2D stealth level.
- By using a non-trivial, Unity 3D-based tool for stealth path-finding, we demonstrate that our approach is effective, and show how choice of different observer agents affects game difficulty.

2. BACKGROUND & RELATED WORK

Stealth games, such as *Mark of the Ninja* and the *Metal Gear Solid* series, are distinguished from games in other genres in that the main game mechanic is to traverse an area undetected by various fixed and/or mobile entities. In such games the potential for detection may be further complicated and/or aided by the presence of different static elements such as shadows and other hiding places, noisy regions, and so forth. Smith defines a level to be *stealth friendly* if the in-game tools that reduce the probability of detection are greater in some respect than the tools that increase that probability [6]. The problem, however, is fundamentally one of avoiding being seen, and thus the task of generating stealth levels combines problems in geometric visibility along with procedural generation.

Visibility - A stealth problem is created by populating a geometric space with enemy agents, either fixed in position, such as (rotating) cameras, or mobile, such as guards, each of which has some well-defined field of view (FOV). Placement of such entities should still permit a solution to exist of course, and we can view the core problem as one of finding a distribution of n *guards* and m *cameras* such that a path exists from *start* to *goal* that does not intersect any enemy FOV. Previous work has investigated the problem of finding such a solution as a path-finding problem [9]. The problem of creating the initial guard/camera placement, however, is more closely related to *visibility* problems in computational geometry. Klee's "art gallery" problem, for example, is defined by a polygon P of n vertices and edges, wherein the task is to determine the minimum of number of fixed points (360° cameras) in P that can see all of P [4]. Variations on this abstract problem have considered (constrained) mobile guards as well. Stealth games extend this problem, introducing time/movement-models, as well as the additional complexity of ensuring the placement is imper-

fect, admitting a solution, with some degree of challenge for a human player. Erdem *et al.* presented a realistic application of the "art gallery" problem where guards do not have 2π rotational and infinitely long view [3]. Using a discrete world (grid-based) they optimally place cameras that can cover every point of P in less than time t. They reduced the problem to a *Set Coverage Problem* using a special case of integer programming. They were able to reach near optimal solutions mainly due to the grid representation.

Procedural Generation - Many generative or procedurally generated content processes [1] for game levels share a core system: a solution is proposed, evaluated, and based on that evaluation the solution is either kept, evolved or rejected [8]. For example, in the work of Togelius *et al.,* they presented generative methods to evolve race tracks; they measured the quality of a proposed track using neural network-based agents, and then evolved the track until the desired criteria, such as the quality of the track, amount of progress, variation in player progress, and difference between maximum and average speeds were met [7]. Other processes for generating game artefacts exist such as fractal noise generators for height-maps or textures, L-systems for plants *etc.* as well as minimizing or maximizing functions using integer/linear programming.

These approaches have also been applied to level design. Shi and Crawfis presented a design tool that computes metrics on the optimal path a player may find to get through a level, given obstacles and enemy distribution [5]. They considered properties such as the minimum-damage cover, longest path, and standard deviation of cover points. This allowed them to change the distribution of obstacles in the level in order to optimize the output. Dormans presents a more holistic approach, describing a general tool that creates whole story, missions and levels [2]. This is based on a grammar logic that produces levels of different shapes by composing prefabricated environment pieces. Our work also aims at level generation, although we are interested in the specific problem of producing an enemy distribution in the context of stealth gameplay.

3. METHOD

Our approach to the guard/camera placement problem is based on application of a series of Monte-Carlo algorithms, with the overall workflow as shown in Figure 1. In order to place a moving guard or a rotational camera, we first discretize the space into different regions. We then use those regions within unique processes that populate either moving guards or cameras. Note that while we present these as distinct tasks, it is trivial to combine these steps and produce a population of both moving guards and cameras.

Discretization and Regions - Placement of both moving guards and cameras are based on a coarse-grained region decomposition. This ensures a good separation of our entities, and guides the actual definition of positions/movements.

In our prototype work, regions are created based on an initial discretization of the space into (fine-grain) grid cells. This allows us to use simple flood-fill algorithms in further decomposition and analysis. Figure 2 shows the result of this first step, with green cells indicating walkable areas and red indicating obstacles.

Region decomposition is created based on an initial seed location for each region. We randomly choose r non-obstacle cells as starting points, separated by distance d. For flexibility in camera/guard placement, and to also help ensure a level remains solveable, we actually generate more regions than necessary, using $r = 2n - 1$ regions for n guards or cameras. From these initial points we then compute a discrete, constrained Voronoi diagram, flooding out from our centers to find the set of cells closer to each starting point than to any other. Figure 3 shows an example of the result for $r = 5$. Seed points are indicated by the small orange dots, and the distinct regions by different colours.

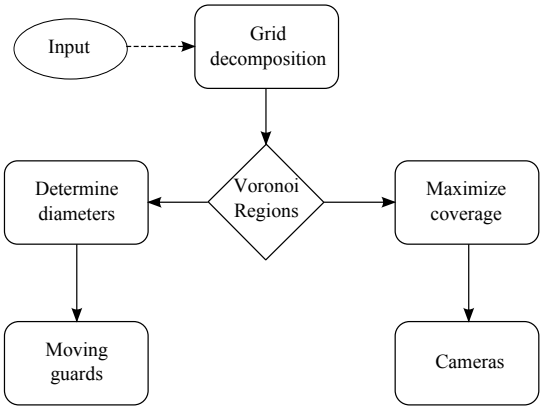

Figure 1: Workflow and main stages.

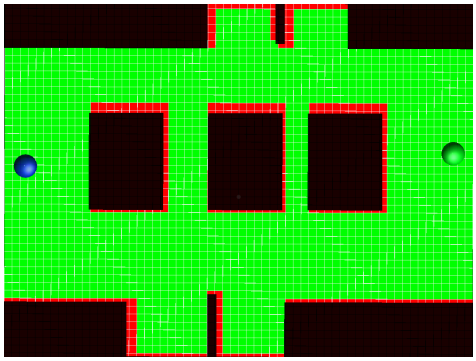

Figure 2: Discretization of the input: walkable in green, unwalkable in red. The green sphere is the *goal* and blue the *start* position. The red cells around obstacles appear offset due to the perspective view of the Unity client.

Moving Guards - Generating arbitrary guard motions is a very difficult task. In our work we restrict guards to patrolling a straight line within a region, with FOVs always facing forward. This is not a complex movement strategy, but even simple straight line patrolling is representative of guard movements found in multiple stealth genre games, including *Mark of the Ninja*. It also constitutes a non-trivial problem, where we need to ensure that patrol paths span the bulk of a region, and so heuristically provide good coverage of that area.

We begin by selecting the n biggest (in area) regions from the r generated regions. Use of distinct regions avoids overlap in guard paths, and thus better coverage. Within each of these n regions we then randomly pick two points until we obtain two points visible to each other in the region. This process is repeated, looking for two points with the largest

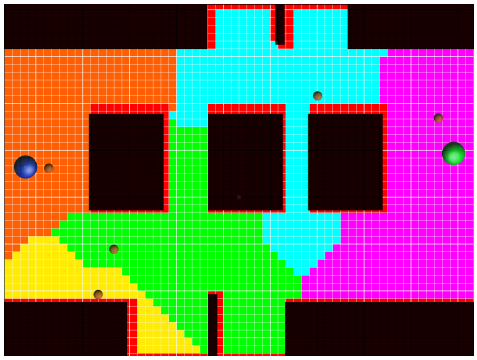

Figure 3: Discrete Voronoi regions constructed from our r seeds.

separation, and so computing an approximate *diameter* for the region. After some iterations, the maximally separated pair of points found is then used to define the patrol path for a guard assigned to that region. Figure 4 shows three guards and their patrol paths based on the same regions shown in Figure 3. Note that we also ensure no patrolling path is looking initially at the start position, and that guards do not continually observe the goal position.

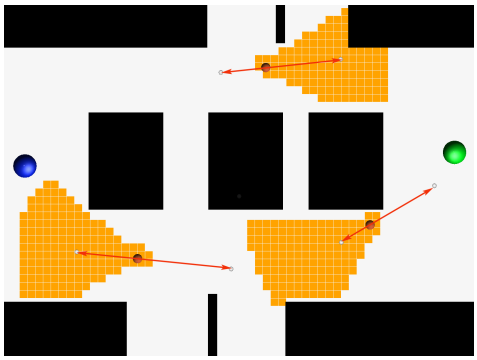

Figure 4: Patrol paths for three moving guards.

Rotational Cameras - As we did for moving guards, we impose specific constraints on camera behaviours to limit the generational complexity. We assume cameras are usually placed on walls, sweeping their FOV back and forth between two angular extremes.

We again begin by choosing the n largest regions. We randomly select a point (cell) along the boundary of the region as a camera location; ideally, this is also an obstacle boundary, although that is not possible in all cases. For each point, we iterate through 8 possible directions (*i.e.,* 45° intervals) and keep track of the two longest lines of sight. This process is repeated some number of times and the candidate pair with the largest total line of sight is kept. The angle formed between these two lines of sight, directed inside the polygonal region, then defines the camera's sweep. Again, we ensure the player's start point is not initially observed, and the goal point not continually observed. Figure 5 shows an example distribution of cameras and their rotational behaviours.

Interface - In order to use this generative process we implemented a tool in Unity 3D. Figure 6 shows part of the

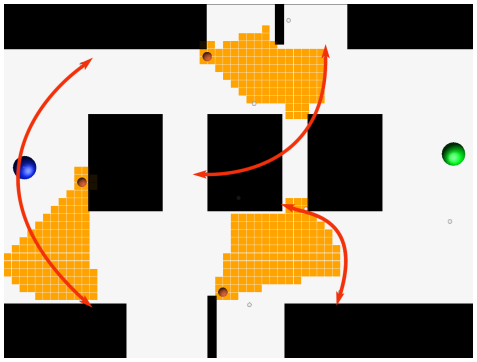

Figure 5: Three cameras and their rotational behaviours.

Figure 6: User interface in Unity 3D.

user interface used for specifying moving guards (camera control is similar).

Through this interface we are able to specify algorithmic parameters, such as the number of moving guards/cameras and iteration thresholds, as well as visualize and control the different algorithmic steps. This implementation design has the further advantage that we are able to integrate it with a Unity tool that computes and visualizes possible stealth paths [9], an example of which is shown in figure 7. This integration lets us analyze our generated output level, determining feasibility and measuring level difficulty.

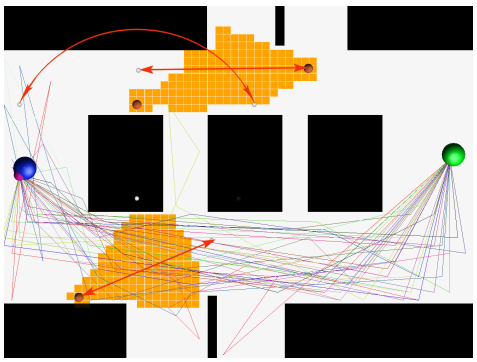

Figure 7: 40 paths generated by Tremblay *et al.*'s stealth solver [9] for 2 guards and 1 camera .

4. EXPERIMENTAL RESULTS

In the previous section we presented a work-in-progress workflow to generate a population of entities (guards and cameras). The process does not show how changing param-

eters influences the game experience. This section discusses preliminary results on measuring difficulty for different distributions of entities.

Intuitively, as the probability of finding a path decreases, the difficulty of a player finding a solution increases. In order to quantitatively measure the level difficulty, we thus looked at the proportion of successful paths found by the randomized solver for a fixed number of searches. Since the solver uses a randomized search, this approximates the likelihood of finding a path in a level.

Here, we used the scene presented in the previous section for analysis. We ran 20 trials where each trial consisted of 20 agents sent to find a path from *start* to *goal* within one unique generated distribution of entities. The relative number of agents that succeed then represents the probability of finding a path. Averages and errors bars from the 20 trials are plotted in Figure 8. We are interested in analysing how difficulty scales with the number of entities, as well as whether the choice of different entity types has any impact.

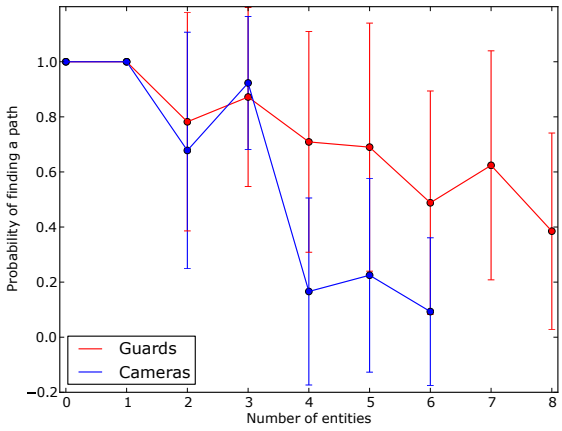

Figure 8: Probability of finding a path as the number of entities increases.

Clearly, the number of moving guards and cameras does influence the ratio of successful paths found. This change is not necessarily linear, however, and despite the high variance in this small study there is a notable increase in difficulty between 3 and 4 cameras. This is likely correlated with the level geometry, where an even distribution of 4 cameras tends to either place cameras near the start or goal positions, or more effectively block the paths between start and goal.

More striking in our results is a strong separation in difficulty between cameras and guards at the same number of entities. This can be understood by considering the different behaviours. Given the same FOV, a patrolling guard will cover a larger area than a sweeping camera, and generally spend less time observing any given spot. Use of mobile guards thus allows players to more easily find opportune times to bypass the guard unseen.

This preliminary experiment shows that cameras and moving guards are meaningfully different and are useful for different design purposes. We hope to build on this, further analyzing the distribution of guards and cameras on multiple game levels in order to better understand how they can be used to control level difficulty and structure the player experience.

5. CONCLUSIONS AND FUTURE WORK

Understanding how guard/camera behaviours and placement affects level design is an important step toward full procedural generation of stealth problems in games. Using our techniques, we can already generate levels with parametrized difficulty. We are especially pleased that despite the strong constraints we rely on in this work, the generated levels actually look quite convincing, suggesting the approach can be both practical and effective.

Our future work is expected to remove many of the implementation constraints in this prototype design, allowing for more flexible geometric analyses, more complex guard/camera behaviours, level structure analysis such as road map, and use of other stealth-affecting game components. We also hope to verify our approach to analysis through human study.

6. ACKNOWLEDGEMENTS

This research was supported by the Fonds de recherche du Québec - Nature et technologies, and the Natural Sciences and Engineering Research Council of Canada. Special recognition to Pedro Andrade Torres and Nir Ricovich.

7. REFERENCES

[1] K. Compton, J. C. Osborn, and M. Mateas. Generative methods. In *The Fourth Procedural Content Generation in Games workshop*, PCG, 2013.

[2] J. Dormans. Adventures in level design: Generating missions and spaces for action adventure games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 1:1–1:8, 2010.

[3] U. M. Erdem and S. Sclaroff. Automated camera layout to satisfy task-specific and floor plan-specific coverage requirements. *Computer Vision and Image Understanding*, 103(3):156 – 169, 2006. Special issue on Omnidirectional Vision and Camera Networks.

[4] J. O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987.

[5] Y. Shi and R. Crawfis. Optimal cover placement against static enemy positions. In *Proceedings of the 8th International Conference on Foundations of Digital Games*, FDG 2013, pages 109–116, 2013.

[6] R. Smith. Level-building for stealth gameplay - Game Developer Conference. `http://www.roningamedeveloper.com/Materials/RandySmith_GDC_2006.ppt`, 2006.

[7] J. Togelius, R. De Nardi, and S. Lucas. Towards automatic personalised content creation for racing games. In *Computational Intelligence and Games*, CIG 2007, pages 252–259, 2007.

[8] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):172–186, 2011.

[9] J. Tremblay, P. A. Torres, N. Rikovitch, and C. Verbrugge. An exploration tool for predicting stealthy behaviour. In *Proceedings of the 2013 AIIDE Workshop on Artificial Intelligence in the Game Design Process*, IDP 2013, pages 34–40, 2013.